Code Based Analysis of Object-Oriented Systems Using Extended Control Flow Graph

A thesis submitted in partial fulfillment for the
degree of Bachelor of Technology

by

Shariq Islam
109CS0324

Under the supervision of

Prof. S. K. Rath

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
**NATIONAL INSTITUTE OF TECHNOLOGY,
ROURKELA, ORISSA, INDIA-769008**

May 2013

Certificate

This is to certify that the thesis entitled "Code Based Analysis of Object-Oriented Systems using Extended Control Flow Graph" submitted by **Shariq Islam**, in the partial fulfillment of the requirements for the award of Bachelor of Technology Degree in Computer Science & Engineering at National Institute of Technology Rourkela is an authentic work carried out by him under my supervision and guidance. To best of my knowledge, The matter embodied in the thesis has not been submitted to any other university / institution for the award of any Degree or Diploma.

Date: Dr. S. K. Rath

Acknowledgements

I am grateful to numerous peers who have contributed toward shaping this project. At the outset, I would like to express my sincere thanks to Prof. S. K. Rath for his advice during my project work. As my supervisor, he has constantly encouraged me to remain focused on achieving my goal. His observation and comments helped me to move forward with investigation depth. He has helped me greatly and always been a source of knowledge.

I am thankful to all my friends. I sincerely thank everyone who has provided me with inspirational words, a welcome ear, new ideas, constructive criticism, and their invaluable time.

I must acknowledge the academic resources that we have acquired from NIT Rourkela. I would like to thank the administrative and technical staff members of the department who have been kind enough to advise and help in their respective roles.

Last but not the least, I would like to dedicate this project to my family, for their love, patience and understanding.

Shariq Islam
Roll:109CS0324

Abstract

The basic features of object-oriented software i.e. encapsulation, inheritance and polymorphism makes it difficult to apply traditional testing methods to them. Traditional testing methods have been successfully implemented in procedural systems. One of the most commonly used example of white-box testing is basis path testing which ensures that every path of a program is executed at least once.

Control Flow Graph(CFG) is a very well-known model that is used for identification of basis paths in procedural systems. McCabes cyclomatic complexity(CC) metric determines that number of linearly independent paths through a piece of software using the control flow graph(CFG) to determine a set of test cases which will cause executable statements to be executed at least once.

The major challenge here is calculation of cyclomatic complexity is easy for procedural systems, but due to basic properties of object-oriented system it is difficult.

My work implements a new model named as Extended Control Flow Graph for code based analysis of object-oriented software. ECFG is a layered CFG where nodes refer to methods rather than statements. My work also implements the calculation of a new metric Extended Cyclomatic Complexity (E-CC).

Contents

List of Figures

Abbreviations

CFG	**C**ontrol **F**low **G**raph
ECFG	**E**xtended **C**ontrol **F**low **G**raph
CC	**C**yclomatic **C**omplexity
E-CC	**E**xtended **C**yclomatic **C**omplexity

Chapter 1

Introduction

1.1 Introduction to Control Flow Graph

A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets. jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves.

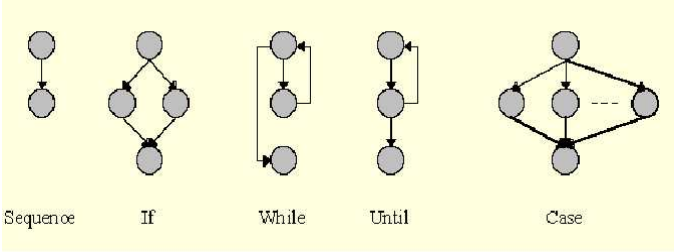


FIGURE 1.1: Examples of CFG

1.1.1 Cyclomatic Complexity of CFG

Cyclomatic complexity (or conditional complexity) is a software metric [1]. It was developed by Thomas J. McCabe, Sr. in 1976 [2] and is used to indicate the complexity of a program. It directly measures the number of linearly independent paths through a program's source code.

Cyclomatic complexity is computed using the control flow graph of the program, The nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program.

The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code. For instance, if the source code contained no decision points such as IF statements or FOR loops, the complexity would be 1, since there is only a single path through the code. If the code had a single IF statement containing a single condition there would be two paths through the code,one path where the IF statement is evaluated as TRUE and one path where the IF statement is evaluated as FALSE.

There are various methods to compute cyclomatic complexity for a given CFG [3] but most commonly used is:

$$CC = E - N + 2$$

1.2 Use of Control flow graph in software testing

Application of cyclomatic complexity is in determining the number of test cases that are necessary to achieve thorough test coverage [4] of a particular module [5] . It is useful because of two properties of the cyclomatic complexity, M, for a specific module:

- M is an upper bound for the number of test cases that are necessary to achieve a complete branch coverage.
- M is a lower bound for the number of paths through the control flow graph (CFG). Assuming each test case takes one path, the number of cases needed to achieve path coverage is equal to the number of paths that can actually be taken. But some paths may be impossible, so although the number of paths through the CFG is clearly an upper bound on the number of test cases needed for path coverage, this latter number is sometimes less than M.

Chapter 2

Literature Review

2.1 Concept of Extended Control Flow Graph

Extended Control Flow Graph is(ECFG) is a new model proposed by S. Bhattacharya and A. Kanjilal in a paper[6][7] it is analogous to CFG for an object-oriented system. It is a layered graphical model representing a collection of CFGs of the individual methods of the OO software. The methods form the nodes of the graph and the edges are drawn if a method calls another method. Each method in itself is similar to a procedural program and has its own CFG depicting the flow of control between statements of a method. Thus every node may further be referring to another graphs. Essentially ECFG has two layers

- The topmost layer represents the methods of individual classes.
- The next layer represents the CFGs of these methods.

Following are the features of ECFG:

1. The graphs is similar to CFG consisting of nodes and edges between nodes except at the topmost level where some nodes may be disconnected. It is a series of graphs arranged in layers.
2. Nodes in CFG refer to statement(s) whereas in ECFG nodes refer to methods.
3. Every methods has associated graphs (CFG) and cyclomatic complexity values. methods, not found in the required class may be a part of its parent class.
4. Object declaration is similar to variable declaration of procedural languages but is not a sequence statement since it refers to constructor method.

5. Edges between nodes are formed whenever any method calls another mehtod. There may be different ways in which nodes are connected.

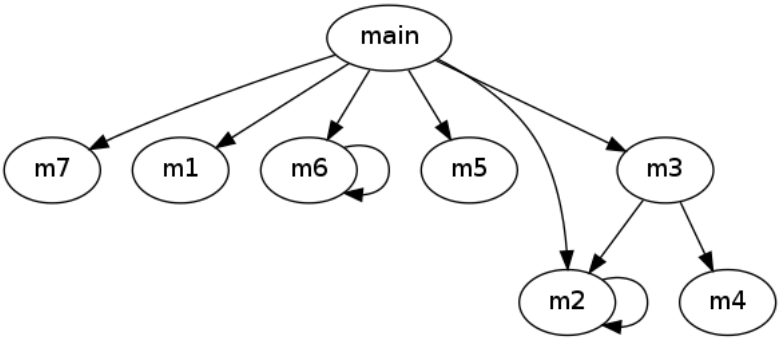


FIGURE 2.1: Example of ECFG

2.2 Concept of Extended Cyclomatic Complexity

In an ECFG, the methods (or CFGs) may be connected in one of the six possible ways. E-CC, the composite complexity of the two or more graphs taken together is distinct in each case. The physical significance of CC i.e. the number of independent paths is maintained while calculating E-CC as well.

The various cases and the E-CC value is as follows and their corresponding proofs are present in the referred paper[7]

Case 1: When two or more graphs are connected in series, i.e. the methods execute in sequence one after the other. If V(G1),V(G2).....V(Gn) are complexities of individual graphs then the E-CC is the largest among them all.

$$E - CC = V(Gx) \;\; if \;\; V(Gx) > V(G1), V(G2).....V(Gn) \;\; and \;\; 1 < x < n$$

Case 2: When two or more graphs are embedded within a graph i.e. a method calls another method which in turn calls another and so on. If V(G1), V(G2) ..... V(Gn) are complexities of each individual graphs and they are embedded within each other i.e. G1 embeds G2, G2 embeds G3 and so on, then the E- CC is given as follows :

$$E - CC = V(G1) + V(G2) + V(G3) + ...V(Gn)(n-1)$$

where n-1 graphs (2 to n) are embedded within G1.The same holds true even if there is no nested embedding i.e. G1 embeds G2,G3... Gn.

Case 3: If a graph embeds another graph more than once, e.g G1 embeds G2 thrice,

then G2 is taken to be embedded only once since once tested it need not be tested again. Composite complexity E-CC will be same as in Case 2 where all the complexity values refer to unique graphs - G1, G2 ..... Gn.

Case 4: When many graphs embed the same graph i.e. more than one method call the same method. Here also the repeated graph is considered only once but the point of entry should be tested in every context.

$$E - CC = (V(G1) + V(G2) - 1) + V(G3) + ... + V(Gn) + (n - 2)$$

where V(G1) : complexity of G(1) that is embedded multiple times and V(G2) ...V(Gn) are complexity values for graphs which embed G1 and n-2 : no. of graphs embedding G1 except one.

Case 5: When one graph is recursively repeated i.e. a method is recursively called.

$$E - CC = V(G1) + 2$$

where V(G1) is the complexity of the method in recursion

Case 6: When recursion involves more than one method. This is a combination of case 1,2,3. Composite complexity

$$E - CC = V(G') + 2$$

where G′ is more than one method connected as per case 1 or 2.

Chapter 3

Proposed Work

3.1 Overview

My proposed work is to automate the process of construction of the new model i.e ECFG[6][7] for JAVA Programs. And also implement the method for calculation of E-CC for object-oriented java program using the constructed ECFG.

The various steps involved are:

- Implementation of a way to parse a java class and construct CFG for given method.
- Traversing the CFG to find Call Nodes and connect Edges between Nodes of ECFG (remember in ECFG methods are the nodes).
- Implement the calculation of E-CC by using six defined cases.

3.2 Construction of CFG for a method

I have used compiled classes for the purpose of analysis due to following reasons

- If source will be used there will be need for parsing and error checking which need not be done for compiled classes
- For source file there can be various outedges for a node example if(a <b && a>0) but for class files there will be at maximum two outedges because class files contains instruction like machine language where a node either evaluates to either true or false.

For construction of CFG for method an API is used known as ASM. ASM is a bytecode engineering library for JAVA. it can be used for control flow analysis using compiled classes.

The overall structure of a compiled class is quite simple Indeed, unlike natively compiled applications, a compiled class retains the structural information and almost all the symbols from the source code. Figure3.1 summarize the overall structure of a compiled class.

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

FIGURE 3.1: Overall structure of a compiled class (* means zero or more)

The ASM tree API reads a class file into hierarchically arranged objects i.e ClassNode Containing a list of MethodNode for methods of class and each MethodNode containing list of InstructionNode.

By using Analyzer class from tree API we can get control flow edges between instructions, so by using Analyzer I saved edges into MethodFlowgraph Object.

A MethodFlowGraph is the actual implementation of CFG for a method it contains

- An array of nodes corresponding to each instruction.
- An adjacency matrix for the graph.

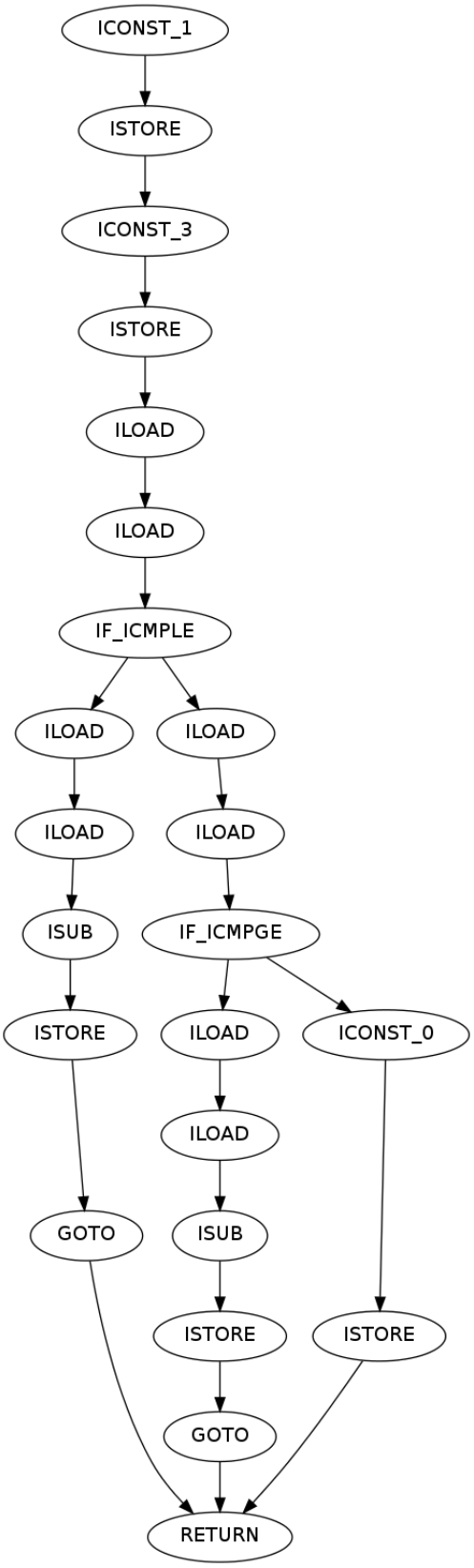


FIGURE 3.2: Example of CFG generated from the program

3.3 Construction of ECFG

The ECFG is constructed by starting from the CFG of main method and adding CFG for called methods recursively.

Let mainCFG be the main method's CFG that can be constructed by supplying the class file to new MethodFlowGraph Object. The ECFG can be constructed by passing this mainCFG to the following algorithm in Figure 3.3

.

```
addGraphsREcursively(MethodFlowGraph : x)
  for each node i in x
    if(I is callNode)
      find the class of called method;
      search for classnode for calledmethods class in cache
      if(notfound)
        create new Classnode for calledmethods class and put in cache;
      else
        c=Called methods classnode from cache;
        search for methodflowgraph in methodflowgraph's cache;
        if(notfound)
          m=create new methodflowgraph for the called method;
          add m to methodfflowgraph cache and nodes of ECFG;
          add edge from x to m;
        else
          m=methodflowgraph from the cache;
          add edge from x to m ;
        endif
      endif
    endif
  endloop
```

FIGURE 3.3: The Algorithm for Constructig ECFG

Explaination:

The Algorithm works as follows:

- It traverses the passed methodflowgraph sequentially instruction by instruction in search of call nodes.

- when a call node is found it finds the class of the called method if it is found in the cache the classnode is taken from cache, otherwise new classnode object is formed and inserted in cache.
- methodsCFG is searched in cache if it is previously made it will be found in cache.if found in cache a edge is added from called graph to the graph from cache, otherwise new methodflowgraph is made and inserted in cache.

3.4 Calculation of E-CC

The extended Cyclomatic Complexity of the ECFG is calculated by following the rules as discussed previously in section 2.2 by traversing the ECFG by following algorithm.

```
int getExtendedCyclomaticComplexity()
    extracalls=0;
    return (calExtendedCyclomaticComplexity(mvMainGraph,lvExploredGraphs)
   +extracalls);


int calExtendedCyclomaticComplexity(MethodFlowGraph x,ExploredGraphs g)
    value=x.getCyclomaticComplexity();
    calledGraphs=Edges.get(pMethodFlowGraph);
if(CalledGraphs isNotEmpty)
    int prevvalue=0;
    For each CalledMethod m
        thisvalue=0;
        if(x isEqualto CalledMethod)) // Case of recursion
            thisvalue=2;
            if(thisvalue>prevvalue)
                prevvalue=thisvalue;
                continue;
            endif
        else if(calledMethod is present in g)// Case 4
            extracalls++;
            continue;
         else
            add CalledMethod to ExploredGraphs g
            thisvalue=calExtendedCyclomaticComplexity(calledMethod,g);
            if(thisvalue>prevvalue)
                prevvalue=thisvalue;
            endif
        endif
    endloop
    value+=prevvalue;
endif
return value;
```

FIGURE 3.4: Algorithm to compute Extended Cyclomatic Complexity (E-CC)

3.5 GUI

The GUI is made using Swing library in Java. Swing is the primary Java GUI widget toolkit. It is part of Oracle's Java Foundation Classes (JFC) an API for providing a graphical user interface (GUI) for Java programs.

It provides various widgets which can be integrated to form a asthetic GUI Application.

The various components used are:

- **JPanel(for holding all the components)**
  JPanel is a public java swing class which is used to create a general-purpose container JPanel objects that are used to group other GUI components without adding any functionality to the added components.

  JPanel panel objects can add color to their background and also can be customized. JPanel inherits methods from its super classes namely JComponent, Container, and Component java classes. JPanel is an extension of java swing JComponent class. JPanel class implements Accessible interface.

- **JLabel(for displaying graph image)**
  JLabel is a Java Swing component that is able to hold text, an icon, or both.

- **JList(for displaying the list of methods)**
  A component that displays a list of objects and allows the user to select one or more items.

- **JSplitPane(used as container for JList and graph image label):**
  JSplitPane is a java swing component used to divide two (and only two) Components. The two Components are graphically divided based on the look and feel implementation, and the two Components can then be interactively resized by the user.

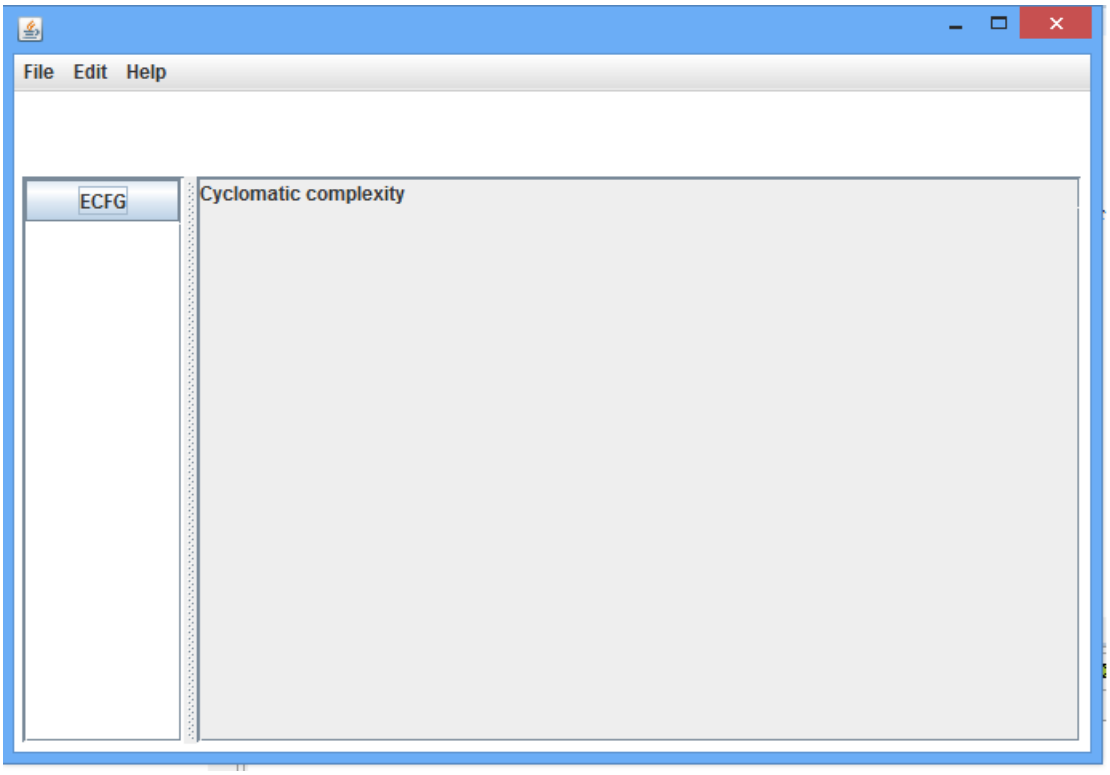


FIGURE 3.5: The GUI of the Program

Chapter 4

Results & Discussions

4.1 Example 1

This is simple program we check for all types of cases of connectivity

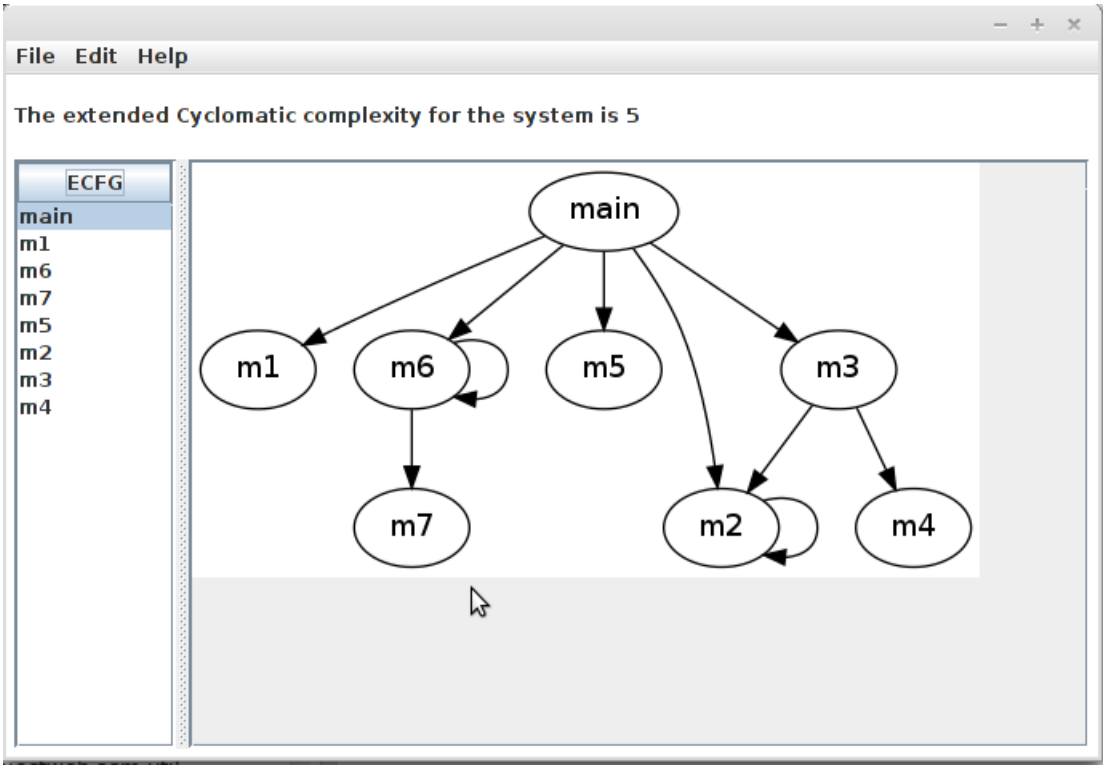
4.1.1 Input Code

```
package test;

public class Test1 {


  static void m1(){
    int a=1;
    int b=2;
    int c;
    if(a>b)
      c=a+b;
    else
      c=0;
  }
  static void m2(){
    m2();
  }
  static void m3(){
    m2();
    m4();
  }
  static void m4(){

  }
```

```
static void m5(){
    int a=1;
    int b=3;
    int c;

    if(a>b)
      c=a-b;
    else if(a<b)
      c=b-a;
      else
        c=0;

  }
  static void m6(){
    m6();
    m7();
  }
   static void m7(){

  }
  /**
   * @param args
   */

  public static void main(String[] args) {
    // TODO Auto-generated method stub
        m1();
    m6();
    m5();
        m2();
        m3();
    //m5();
  }

}
```

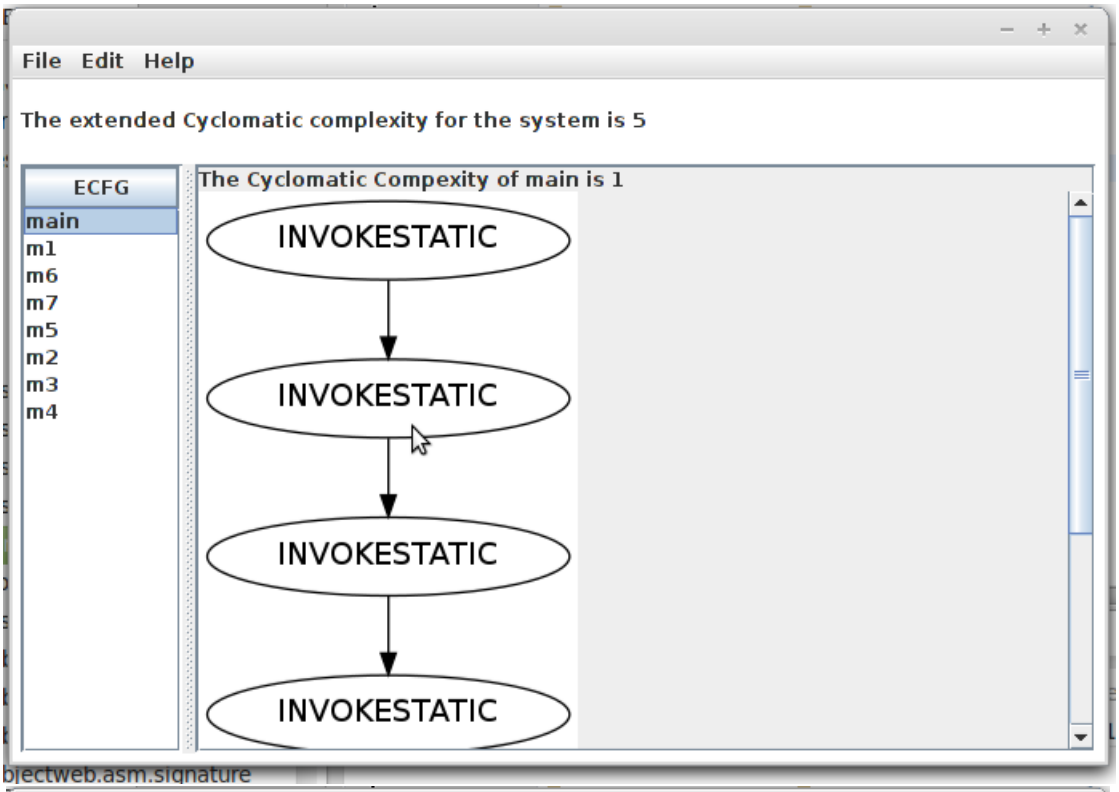
Listing 4.1: Input Source Code

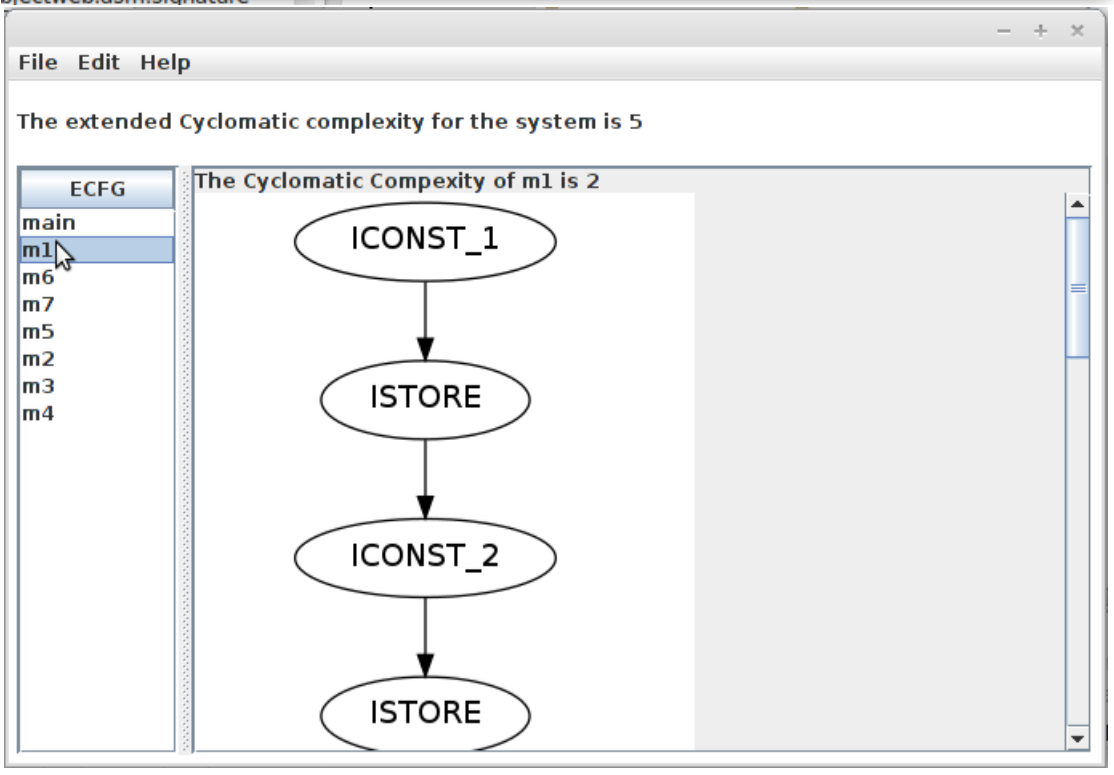
4.1.2 Output

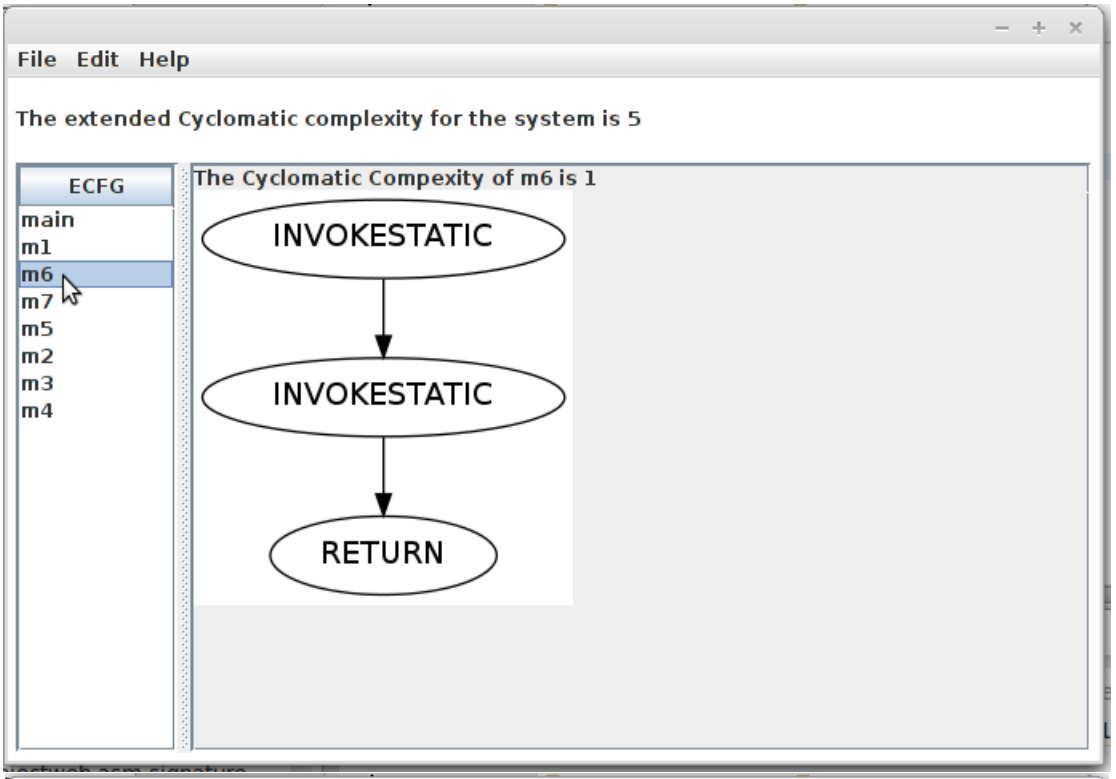


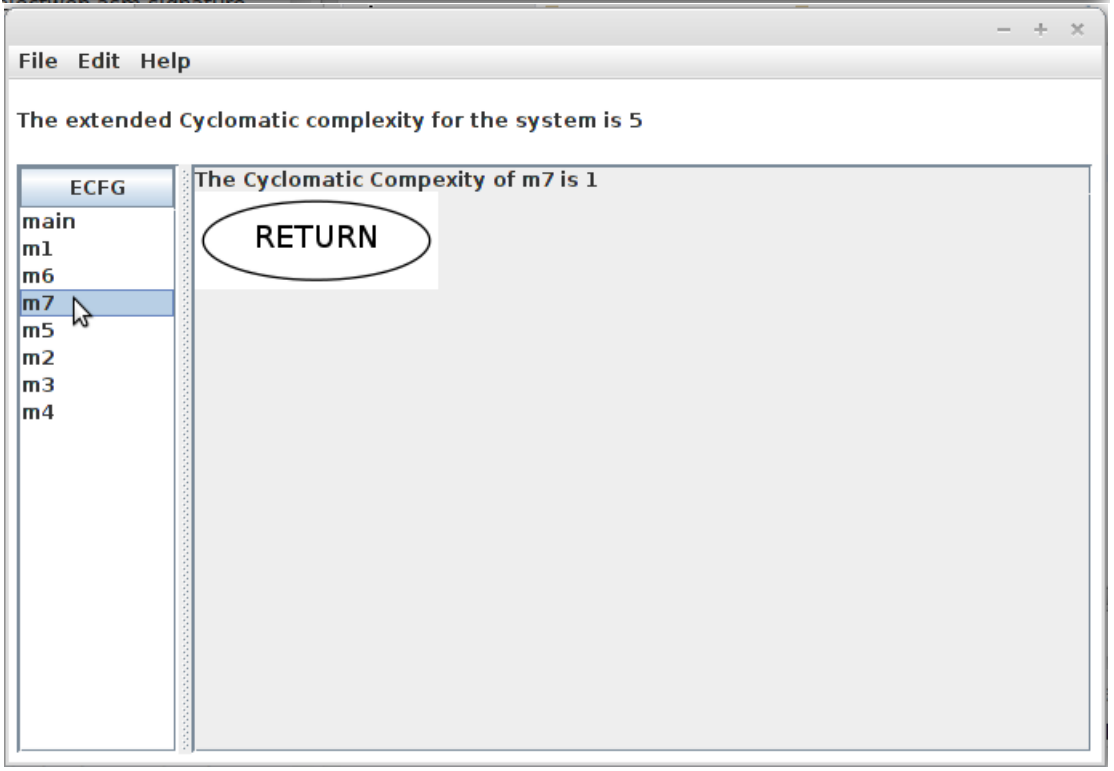
ECFG For The Input Code Above.

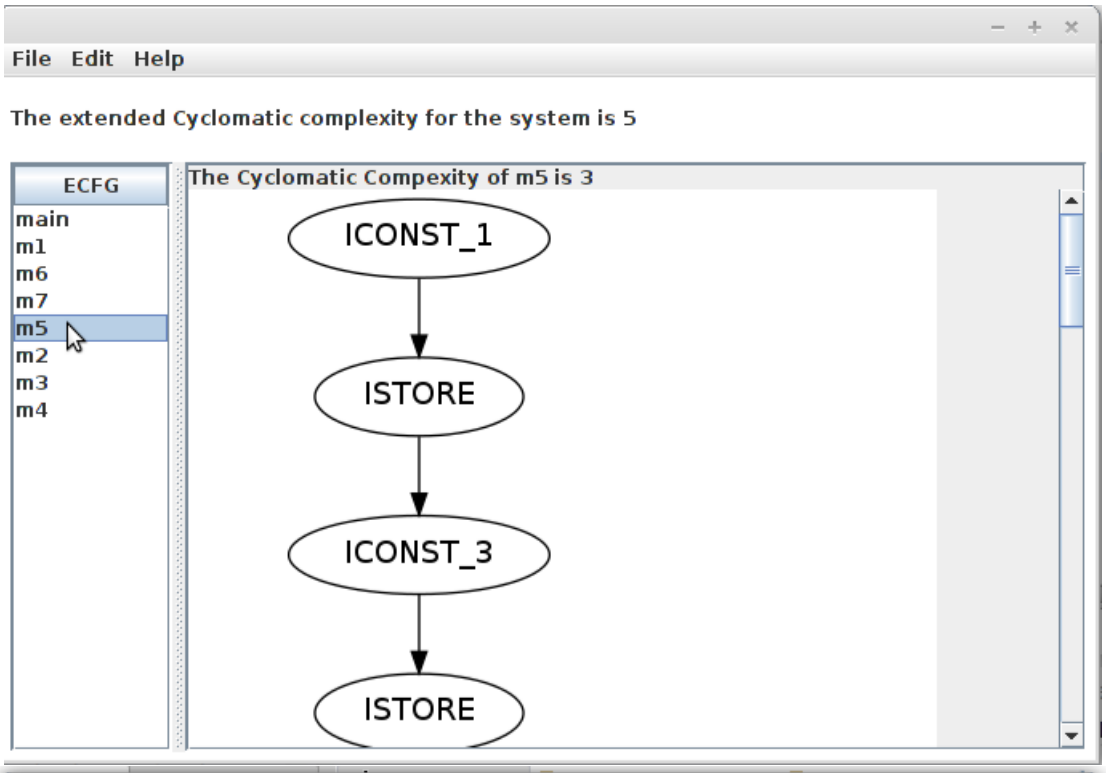
The above image shows the snapshot of the software showing ECFG and the E-CC of the system. The individual CFG's (some of them are not completely visible due to limitations of page)of methods and their CC are as follows.

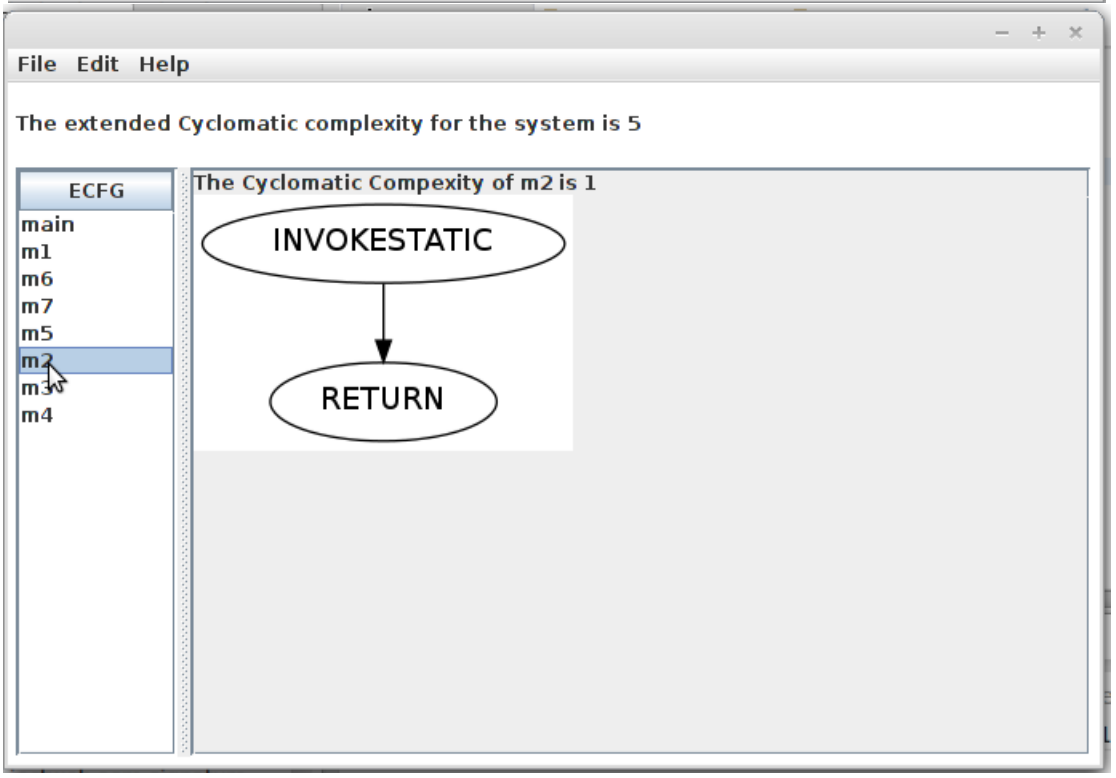

File Edit Help
The extended Cyclomatic complexity for the system is 5
ECFG
main
m1
m6
m7
m5
m2
m3
m4
The Cyclomatic Compexity of main is 1
INVOKESTATIC
INVOKESTATIC
INVOKESTATIC
INVOKESTATIC

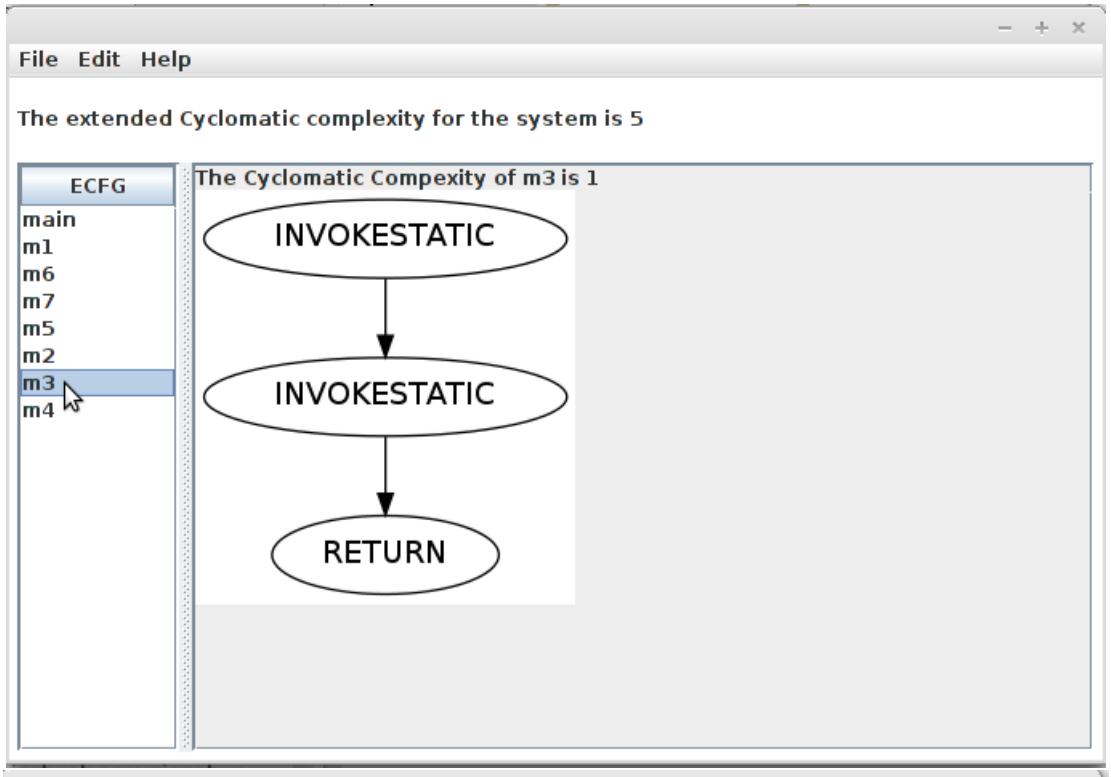

File Edit Help
The extended Cyclomatic complexity for the system is 5
ECFG
main
m1
m6
m7
m5
m2
m3
m4
The Cyclomatic Compexity of m1 is 2
ICONST_1
ISTORE
ICONST_2
ISTORE

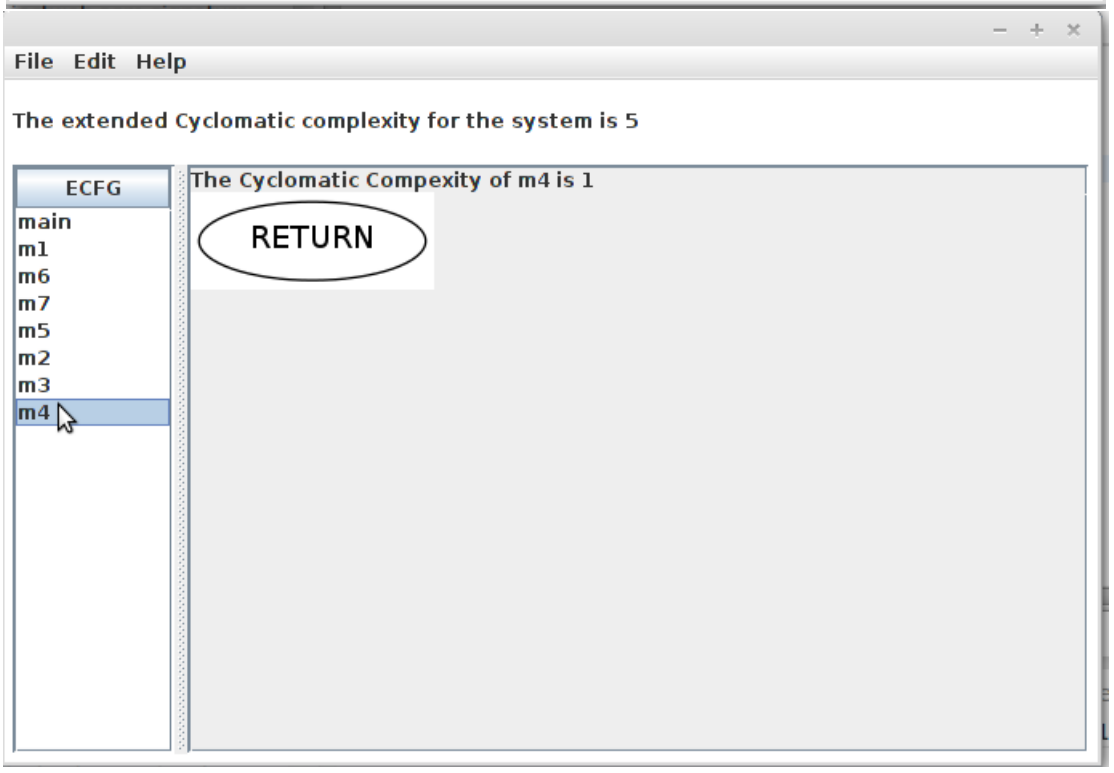

File Edit Help
The extended Cyclomatic complexity for the system is 5
ECFG
main
m1
m6
m7
m5
m2
m3
m4
The Cyclomatic Compexity of m6 is 1
INVOKESTATIC
INVOKESTATIC
RETURN


File Edit Help
The extended Cyclomatic complexity for the system is 5
ECFG
main
m1
m6
m7
m5
m2
m3
m4
The Cyclomatic Compexity of m7 is 1
RETURN


File Edit Help
The extended Cyclomatic complexity for the system is 5
ECFG
main
m1
m6
m7
m5
m2
m3
m4
The Cyclomatic Compexity of m5 is 3
ICONST_1
ISTORE
ICONST_3
ISTORE


File Edit Help
The extended Cyclomatic complexity for the system is 5
ECFG
main
m1
m6
m7
m5
m2
m3
m4
The Cyclomatic Compexity of m2 is 1
INVOKESTATIC
RETURN


File Edit Help
The extended Cyclomatic complexity for the system is 5
ECFG
main
m1
m6
m7
m5
m2
m3
m4
The Cyclomatic Compexity of m3 is 1
INVOKESTATIC
INVOKESTATIC
RETURN


File Edit Help
The extended Cyclomatic complexity for the system is 5
ECFG
main
m1
m6
m7
m5
m2
m3
m4
The Cyclomatic Compexity of m4 is 1
RETURN

4.2 Example 2

This is a real life example. I have taken a small ATM Module [8] for testing.

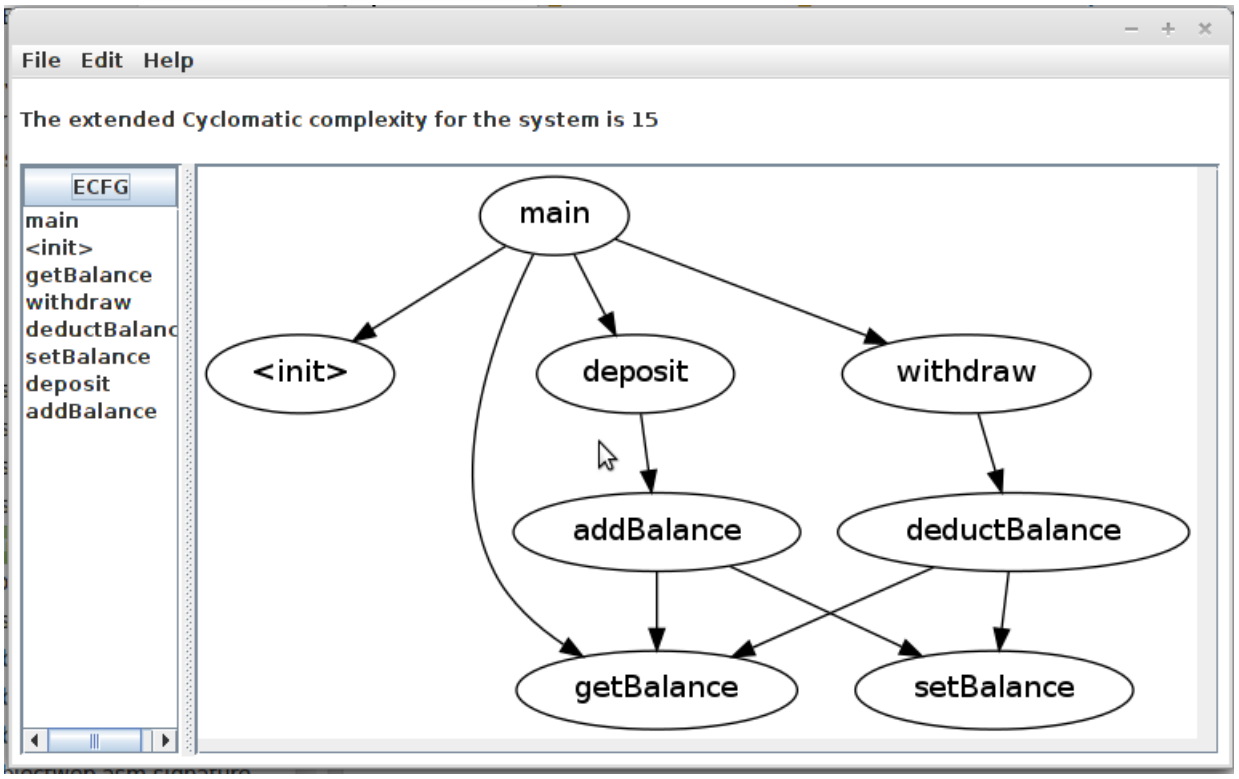
4.2.1 Input Code

```
package test;

import java.util.Scanner;

public class ATM {

  private int balance;
   int getBalance(){
    return balance;
  }

    void withdraw(int amount){
    if(balance<1000)
      System.out.println("insufficient funds to withdraw");
    else if(balance-amount<1000)
      System.out.println("balance after the withdrawl should not be less
   than 1000");
    else
      deductBalance(amount);
  }
   void deposit(int amount){
    addBalance(amount);
  }
   void deductBalance(int amount){
    int bal;
    bal=getBalance();
    bal-=amount;
    setBalance(bal);

  }
   void addBalance(int amount){
    int bal=getBalance();
    bal+=amount;
    setBalance(amount);
  }
   void setBalance(int amount){
    balance=amount;
  }
```

```
  /**
   * @param args
   */

  public static void main(String[] args) {
    // TODO Auto-generated method stub
        int choice=0;
        ATM a=new ATM();
        while(choice!=-1){
    System.out.println("Enter your choice");
    System.out.println("1.Check Balance");
    System.out.println("2.Withdraw Amount");
    System.out.println("3.Deposit Money");
    System.out.println("Enter -1 to exit");
    Scanner s=new Scanner(System.in);
    choice=s.nextInt();
    if(choice >3 ||choice <1)
      continue;
    else if(choice==1)
      System.out.println(a.getBalance());
    else if(choice==2){
      System.out.println("Enter Amount to withdraw");
      int amount=s.nextInt();
      a.withdraw(amount);
      }
    else if(choice==3){
      System.out.println("Enter Amount to Deposit");
      int amount=s.nextInt();
      a.deposit(amount);
      }


        }
  }

}
```

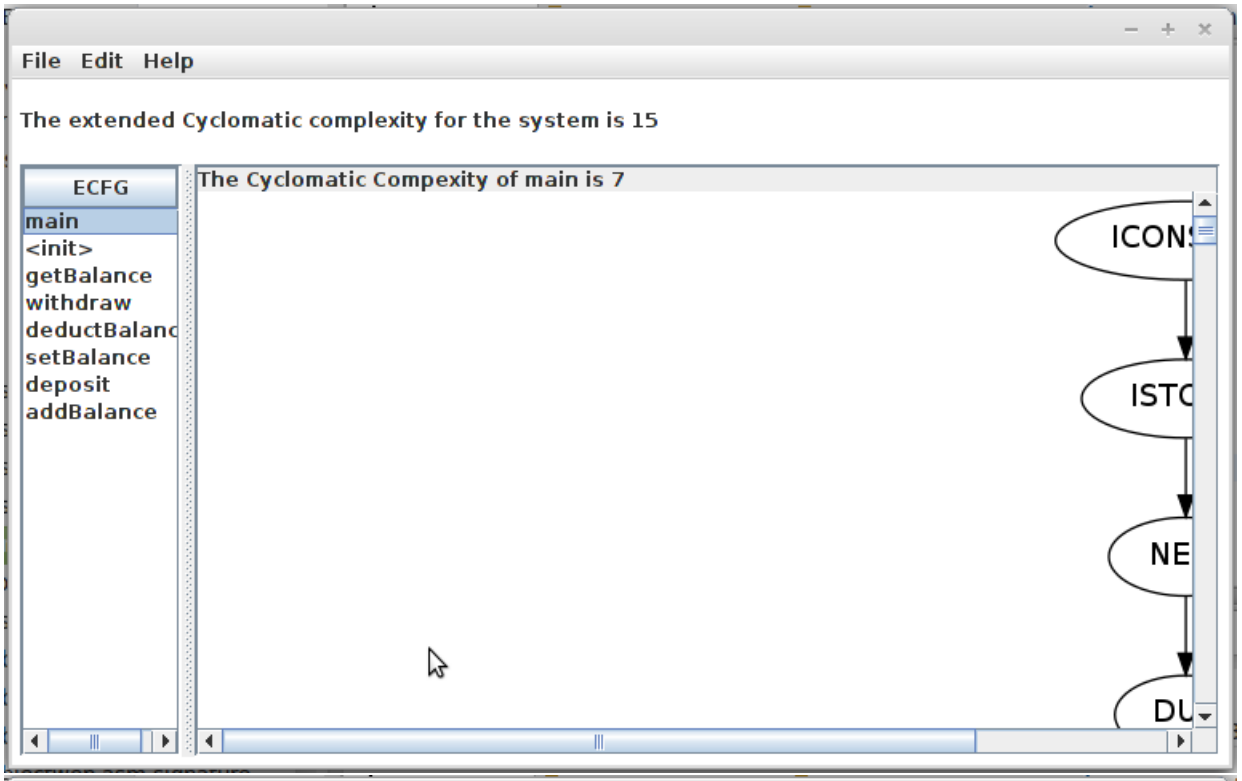
Listing 4.2: Input Source Code

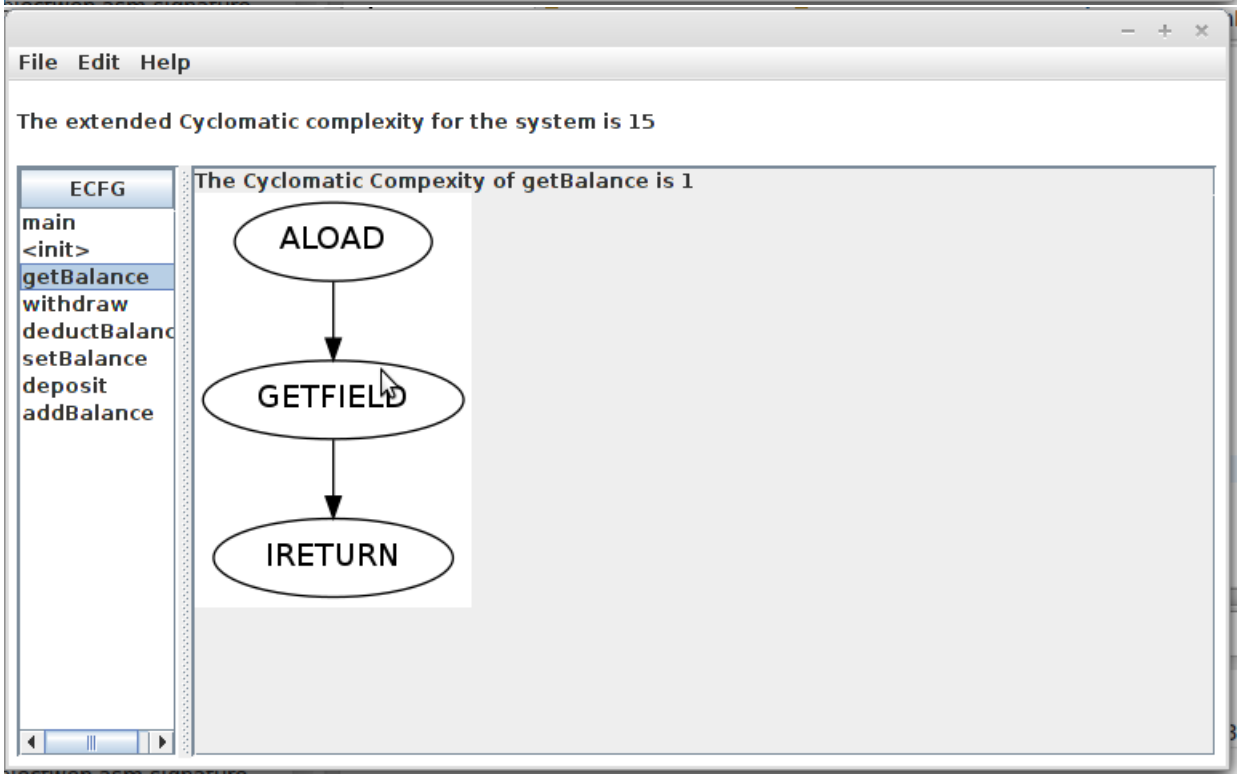
4.2.2 Output

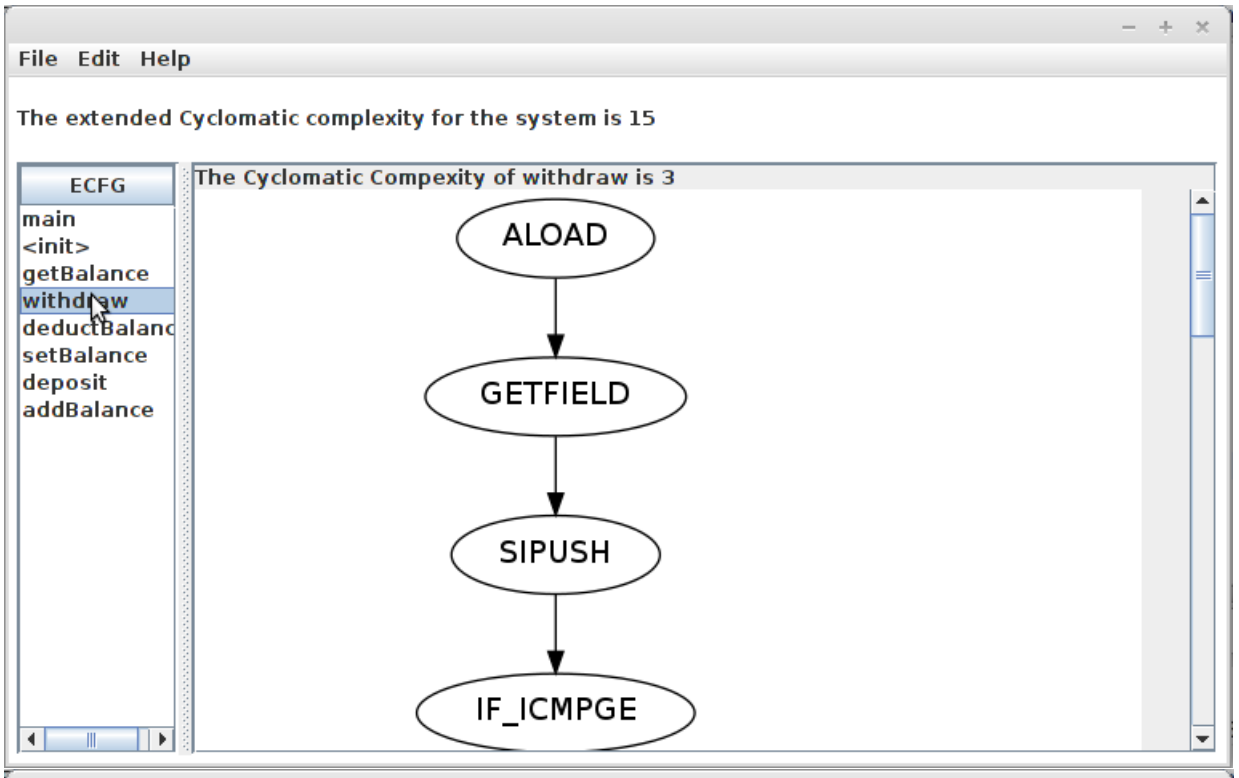


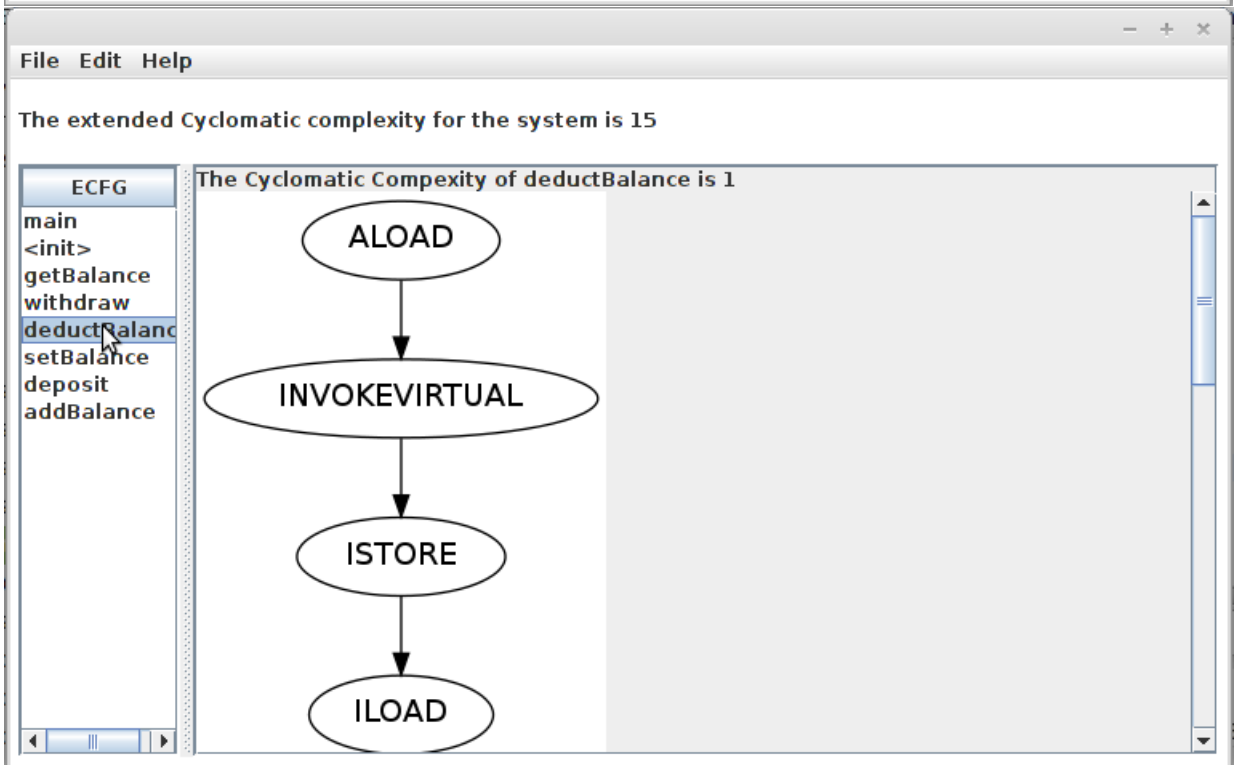
ECFG For The Input Code Above.

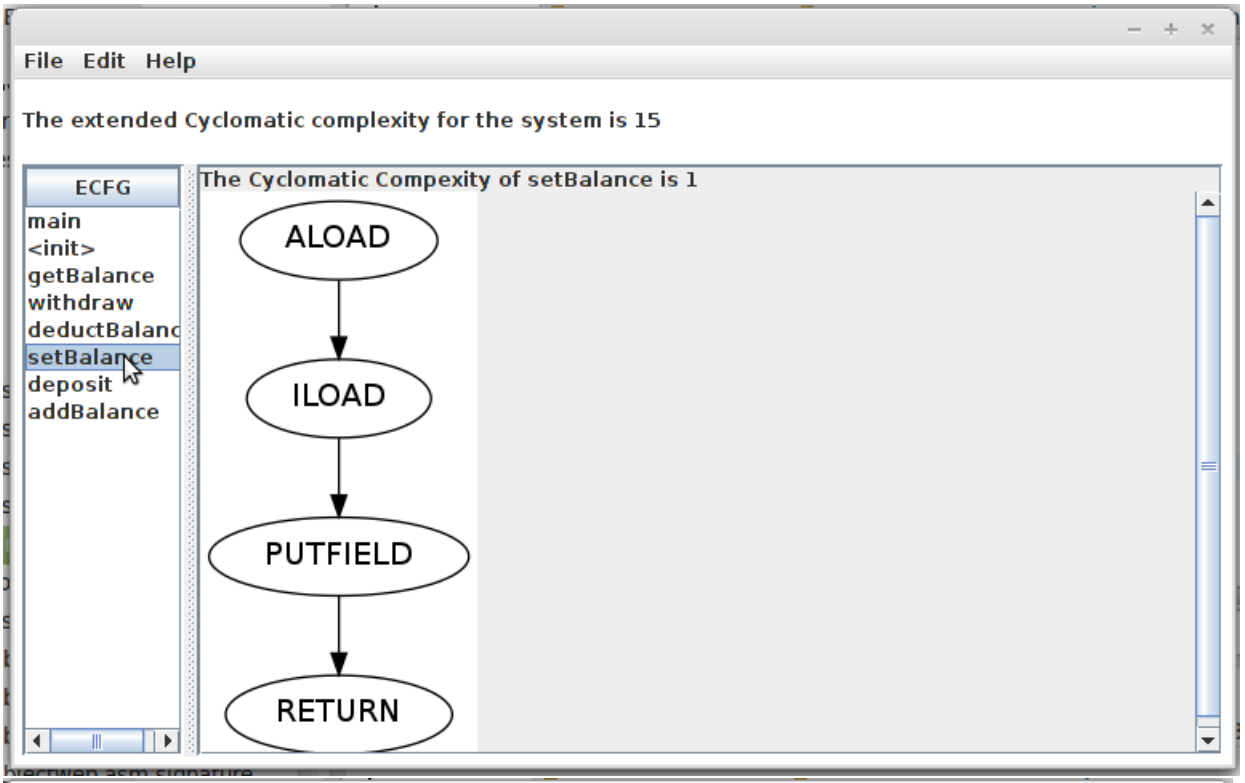
The above image shows the snapshot of the software showing ECFG and the E-CC of the system. The individual CFG's (some of them are not completely visible due to limitations of page) of methods and their CC are as follows.

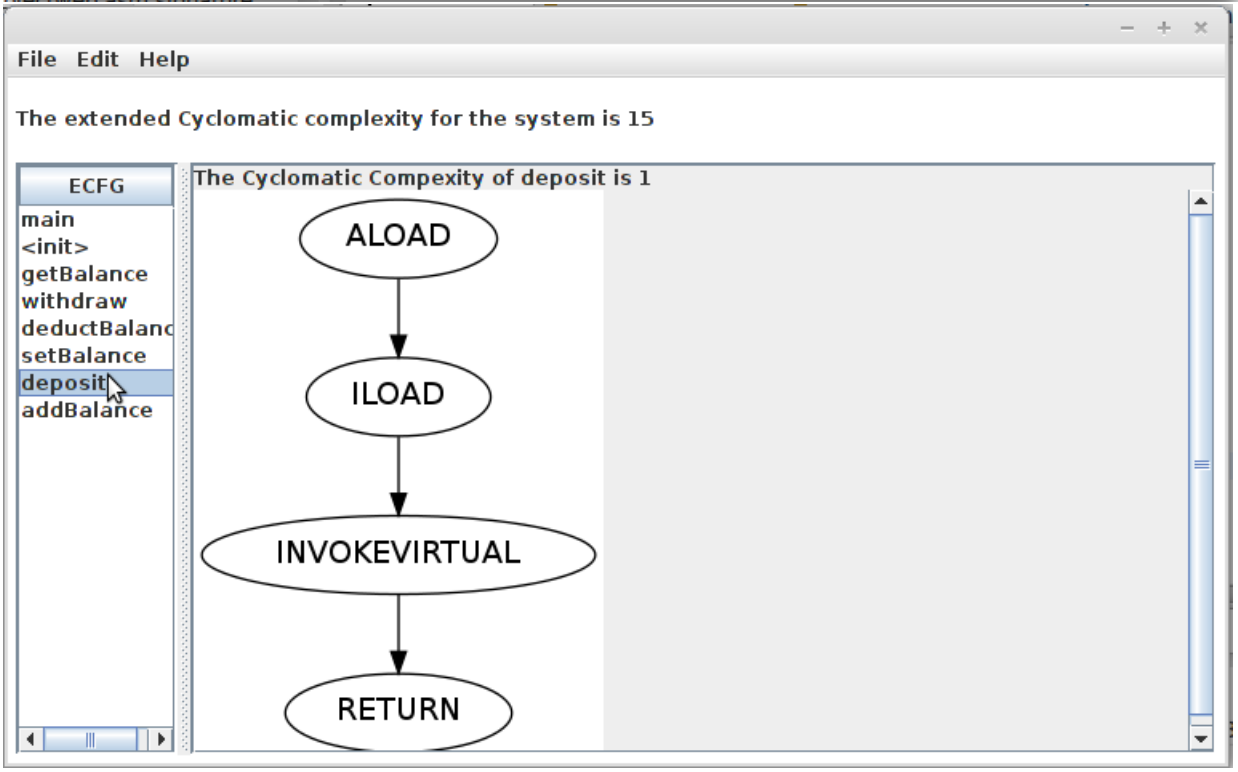

File Edit Help
The extended Cyclomatic complexity for the system is 15
ECFG
main
<init>
getBalance
withdraw
deductBalanc
setBalance
deposit
addBalance
The Cyclomatic Compexity of main is 7
ICON
ISTC
NE
DU

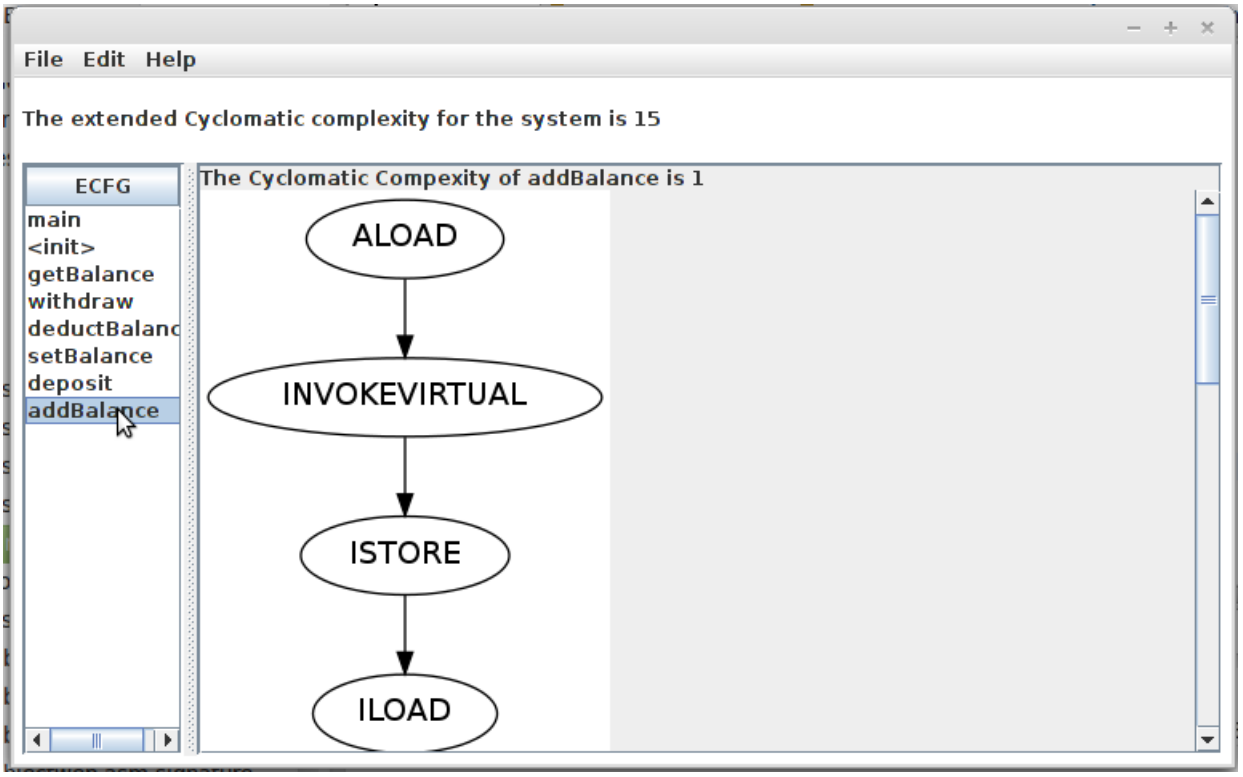

File Edit Help
The extended Cyclomatic complexity for the system is 15
ECFG
main
<init>
getBalance
withdraw
deductBalanc
setBalance
deposit
addBalance
The Cyclomatic Compexity of getBalance is 1
ALOAD
GETFIELD
IRETURN


File Edit Help
The extended Cyclomatic complexity for the system is 15
ECFG
main
<init>
getBalance
withdraw
deductBalanc
setBalance
deposit
addBalance
The Cyclomatic Compexity of withdraw is 3
ALOAD
GETFIELD
SIPUSH
IF_ICMPGE


File Edit Help
The extended Cyclomatic complexity for the system is 15
ECFG
main
<init>
getBalance
withdraw
deductBalanc
setBalance
deposit
addBalance
The Cyclomatic Compexity of deductBalance is 1
ALOAD
INVOKEVIRTUAL
ISTORE
ILOAD


File Edit Help
The extended Cyclomatic complexity for the system is 15
ECFG
main
<init>
getBalance
withdraw
deductBalanc
setBalance
deposit
addBalance
The Cyclomatic Compexity of setBalance is 1
ALOAD
ILOAD
PUTFIELD
RETURN


File Edit Help
The extended Cyclomatic complexity for the system is 15
ECFG
main
<init>
getBalance
withdraw
deductBalanc
setBalance
deposit
addBalance
The Cyclomatic Compexity of deposit is 1
ALOAD
ILOAD
INVOKEVIRTUAL
RETURN


File Edit Help
The extended Cyclomatic complexity for the system is 15
ECFG
main
<init>
getBalance
withdraw
deductBalanc
setBalance
deposit
addBalance
The Cyclomatic Compexity of addBalance is 1
ALOAD
INVOKEVIRTUAL
ISTORE
ILOAD

Chapter 5

Conclusion

In this project we implemented a graph based methodology for analysis of OO systems focusing on the structure of program code and arrives at an analogous model to CFG for testing of OO systems. Cyclomatic Complexity metric identifies the minimum number of paths required for testing. We propose to use the E-CC metric and derive test paths for OO systems similar to McCabe's basis set which would be essential for test vector generation [9] .

Bibliography

[1] N.V. Balasubramanian. Object-orieneted metrics. *Asia-Pacific Software Engineering Conference*, pages 30–34, 1996.

[2] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[3] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, April .

[4] Williams B. S. A survey on Code Coverage as a Stopping Criterion for Unit Testing. *Dept. of Computer Science, North Carolina State University Technical Report TR-2008-22*, 2008.

[5] Glenford J. Myers. *The art of software testing.* Wiley, 2 edition, 2004.

[6] Swapan Bhattacharya and Ananya Kanjilal. Code Based Analysis For Object-Oriented Systems. *J. Comput. Sci. Technol.*, 21(6):965–972, 2006.

[7] Swapan Bhattacharya and Ananya Kanjilal. Static Analysis of Object Oriented Systems Using Extended Control Flow Graph. *IEEE,proceedings of TENCON 2004. 2004 IEEE Region 10 Conference*, pages 310–313, 2004.

[8] Michael R Blaha and James R Rumbaugh. *Object-oriented modeling and design with UML.* Pearson, 2 edition, 2005.

[9] Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.